

# Software Testing

## Inhaltsverzeichnis

Ablauf des Seminars .....	2
Motivation zum Testen .....	3
Kleine Beispiele aus der Praxis .....	3
Kosten von Fehlern .....	3
Getting Things Done .....	4
Broken Window Theorie .....	5
Grundbegriffe .....	6
Grundbegriff Fehler .....	6
Grundbegriff Test .....	6
Grundbegriff Softwarequalität .....	7
Grundbegriff Testaufwand .....	7
Aufgabe .....	7
Testen im Software-Lebenszyklus .....	9
Testplanung und Steuerung .....	9
Testziele .....	9
Teststufen .....	9
Last- und Stresstest .....	10
Statische Tests .....	11
Dynamische Tests .....	11
BlackBox-Verfahren .....	11
WhiteBox-Verfahren .....	11
Test-Driven-Development (TDD) .....	13
Aufgabe .....	14
A. Anhang .....	15
A1. Links & Quellen .....	15
A2. Stichwortverzeichnis .....	16
A3. Abkürzungsverzeichnis .....	16

# Ablauf des Seminars

- Motivation des Testens
- Grundbegriffe (kurz)
- Testen im Software-Lebenszyklus
  - Vorgehensmodelle
  - Teststufen (Komponenten-, Integrations-, System-, Abnahmetest)
  - Last- und Stresstest
- Statische Tests
- Dynamische Tests (Schwerpunktthema)
  - Black-Box-Verfahren
  - White-Box-Verfahren

Es werden Schwerpunkte auf praktische Elemente des Testens gelegt. Die theoretischen Grundlagen sind Fleißarbeit in der Eigenverantwortung der Teilnehmer und können in der einschlägigen Literatur (s. Buchempfehlung) leicht nachgeholt werden.

# Motivation zum Testen

## Kleine Beispiele aus der Praxis

Z.B. unter [https://de.wikipedia.org/wiki/Liste\\_von\\_Programmfehlerbeispielen](https://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen) gibt es schöne Beispiele aus der Praxis welche fatalen Auswirkungen eine ungenügende Testabdeckung haben können. Zwei bekannte Beispiele sind folgende:

### Teurer Fehler bei Float und Integer:

Am 4. Juni 1996 sprengte sich die erste **Ariane-5-Rakete** (Startnummer V88) der Europäischen Raumfahrtbehörde 40 Sekunden nach dem Start in vier Kilometern Höhe automatisch. Der Programmcode für (unter anderem) die Vorstart-Ausrichtung war von der Ariane 4 übernommen worden, lief unnötigerweise auch nach dem Start weiter und funktioniert dabei nur in einem von der Ariane 4 nicht überschreitbaren Bereich der Horizontalgeschwindigkeit. Als dieser Bereich von der Ariane 5 verlassen wurde, da sie höhere Horizontalgeschwindigkeiten erreicht als die Ariane 4, schlug der Fehler auf die Trägheits-Steuersysteme durch und diese schalteten sich weitgehend ab. Bei der Programmierung war es zu einem Fehler bei der Typumwandlung gekommen. Als von Float nach Integer umgewandelt wurde und der **Wert 32.768** erreichte, entstand ein Überlauf. Dieser Überlauf hätte durch die verwendete Programmiersprache Ada eigentlich entdeckt und behandelt werden können. Diese Sicherheitsfunktionalität ließen die Verantwortlichen jedoch abschalten.

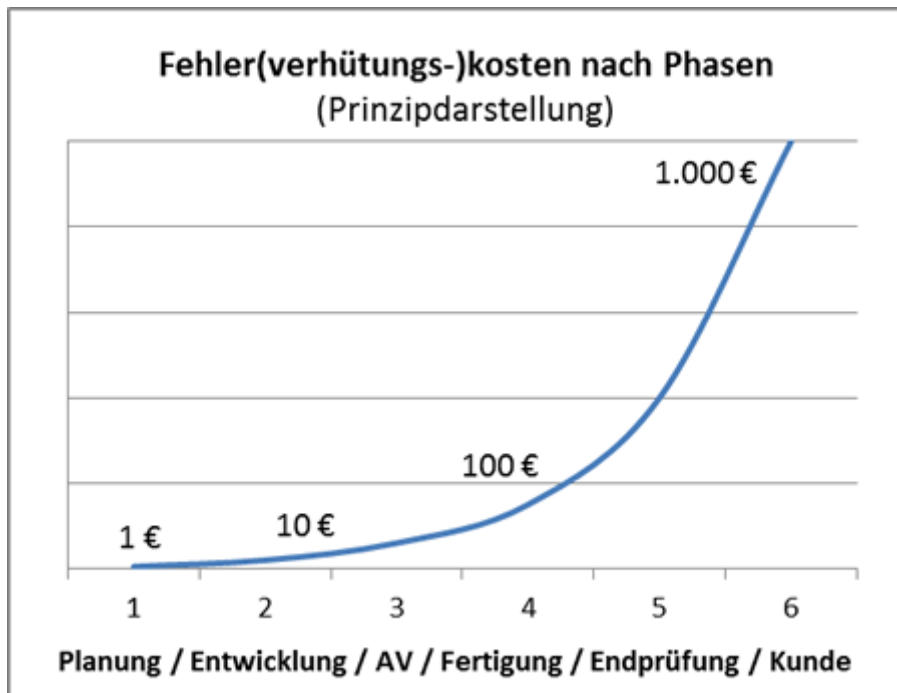
**Der Schaden betrug etwa 370 Millionen US-Dollar.**

### Peinliche Division durch 0

Die USS Yorktown, ein weitgehend computerisiertes Schiff der Ticonderoga-Klasse, trieb nach Aussage der Navy bei einem Manöver im September 1997 knapp drei Stunden hilflos im Mittelmeer, nachdem ein Ingenieur "0" direkt in die Datenbank eingegeben hatte um einen fehlerhaften Eintrag eines Sensors zu korrigieren. Die Software verwendete diesen Wert für Divisionen, die Folge waren „Division durch null“-Fehler, welche von der Software nicht korrekt abgefangen wurden. Es füllte sich im weiteren Verlauf der temporäre Speicher des Systems und als dieser voll war, wurde beim Überlauf der angrenzende Speicherbereich, der des Antriebssystems, überschrieben. Das Antriebssystem und das Rechnernetzwerk, das aus mehreren Windows-NT-Servern bestand, fielen aus, da sich der Fehler im Netzwerk ausgebreitet hatte. Es dauerte über drei Stunden, die Server zu reanimieren.

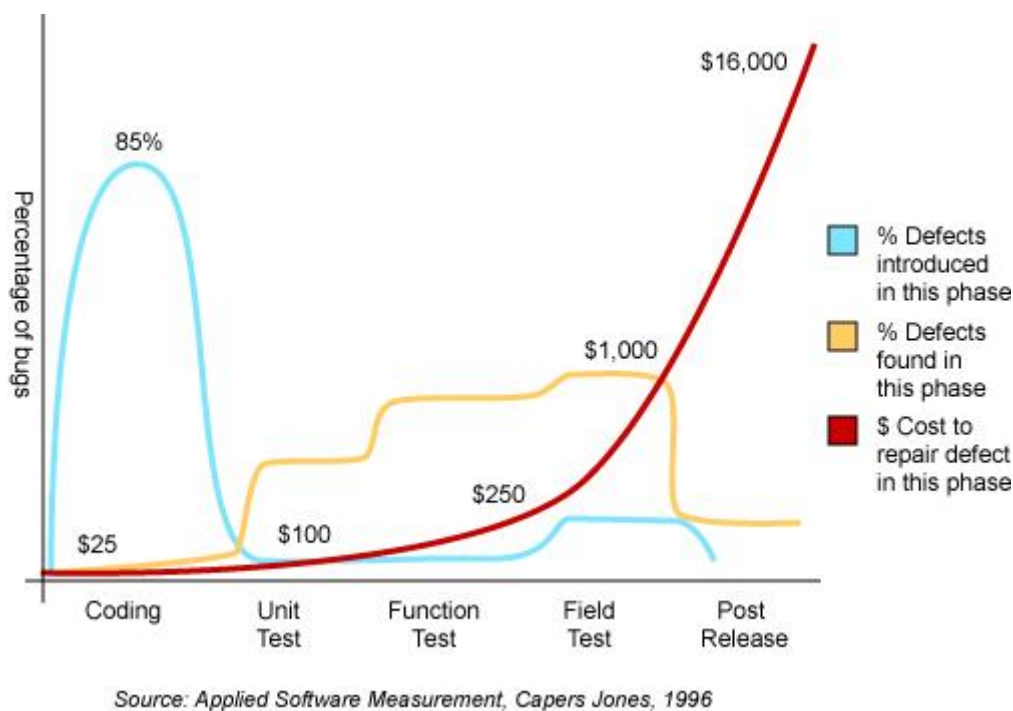
## Kosten von Fehlern

Grundsätzlich gilt, dass ein Fehler teurer wird je später er entdeckt und behoben wird. Einer der Ansätze ist die 10er-Regel:



(Quelle: <http://olev.de/0/10er-regl.htm>)

Bezeichnend ist der exponentielle Anstieg der Kosten was auch nochmal schön in diesem Zusammenhang deutlich wird:



(Quelle: <https://deanondelivery.com/product-managers-do-you-know-how-much-your-bugs-cost-72b6e36e7684>)

## Getting Things Done

- ☒ Ein ganz wichtiger Punkt in der Softwareentwicklung ist der, an dem man eine Aufgabe erledigt hat.
- ☐ Wie definiert man nun diesen Punkt?

□ Wann genau ist eine Aufgabe erledigt?

Hierbei sind Tests ein sehr hilfreiches Mittel. Mit dem Test können wir definieren was das Ziel ist und wir können immer wieder überprüfen ob dieses Ziel auch in Zukunft (nach anderen Änderungen) noch erreicht ist.

"Never touch a running system!" ist ein Ausspruch für schlechtes Testen. Wenn ich eine gute und saubere Testabdeckung habe kann ich gerne ändern. Wenn danach alle meine Tests wieder sauber durchlaufen haben meine Änderungen keine Seiteneffekte. Ich bin frei zu ändern.

### **Praxistip:**

Wenn eine neue Aufgabe begonnen wird, wird hierfür (hoffentlich) von dem Develop-Branch ein neuer Feature-Branch abgezweigt. Bevor man mit der Aufgabe beginnt, sollte man sicherstellen, dass auf diesem Branch alle Tests funktionieren. Dann ist man sich sicher, dass wenn nachher irgendwelche Tests nicht funktionieren man selbst die Ursache war.

1. Feature-Branch erstellen und auschecken
2. neuen Branch testen
3. Änderungen machen
4. Branch testen
5. PullRequest stellen

TDD, wenn möglich.

## **Broken Window Theorie**

Wenn man einmal schlusig mit den Tests ist und die Qualität vernachlässigt, wird die Vernachlässigung immer schlimmer.

⇒ Kein Vertrauen in die Software und die eigene Arbeit

⇒ Keine Wertschätzung in das Produkt

# Grundbegriffe

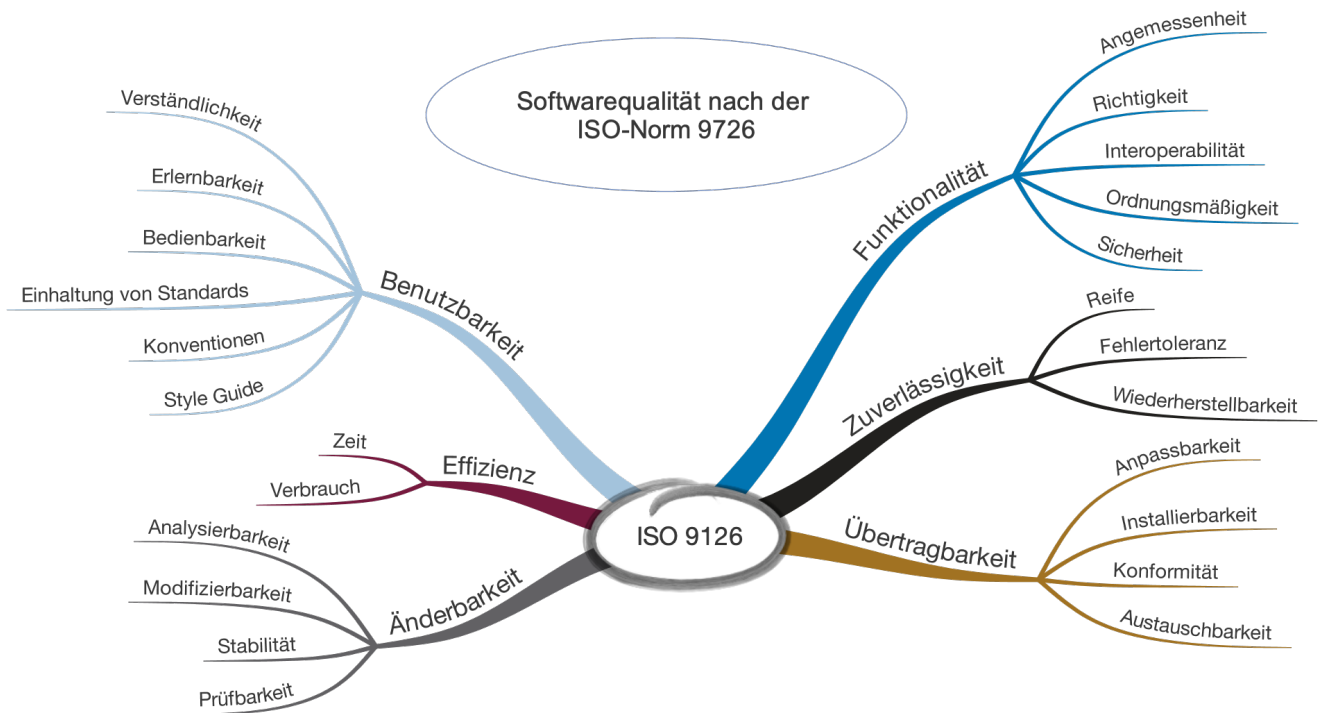
## Grundbegriff Fehler

- Fehlerbegriff
  - Nichterfüllung festgelegter Anforderungen
  - Abweichung von Ist- und Sollverhalten
- Fehlerwirkung
  - Ursprung in Fehlerzustand
- Fehlermaskierung
- Fehlhandlung
  - Programmierfehler
  - Vergessene Anforderung

## Grundbegriff Test

- Testbegriff
  - Debugging
  - Test → Stichprobenhafte Prüfung
- Ziele des Testens
  - Fehlerwirkung nachweisen
  - Qualität bestimmen
  - Vertrauen in das Programm erhöhen
  - Analysieren des Programms od. der Dokumente zur Vorbeugung von Fehlerwirkungen

# Grundbegriff Softwarequalität



## Grundbegriff Testaufwand

- Testaufwand
  - Fehlerfreiheit nicht nachweisbar
  - Vollständiger Test nicht durchführbar
  - Zu viele kombinierbare Möglichkeiten
- Einflussfaktoren für die Bemessung des Testaufwandes
  - Durch Fehler verursachte Kosten können in die Millionen gehen
  - Risiko bestimmt sich aus Eintrittswahrscheinlichkeit und zu erwartender Schadenshöhe
  - Es kann Gefahr für Leib oder gar Leben bestehen
  - Testaufwand in Abhängigkeit von Risiko und Gefährdungsbewertung
  - Kombination aus unterschiedlichen Testverfahren notwendig
  - Testfallexplosion

## Aufgabe

Wie wird wohl der Testaufwand bei folgenden Projekten aussehen?

- ☐ Banking-App der BNP Paribas-Bank
- ☐ Parkplatz-Scharing für Mitarbeiter einer Firma
- ☐ Steuerungssoftware für Langstreckenraketen
- ☐ Steuerung für Beatmungssysteme im medizinischen Bereich

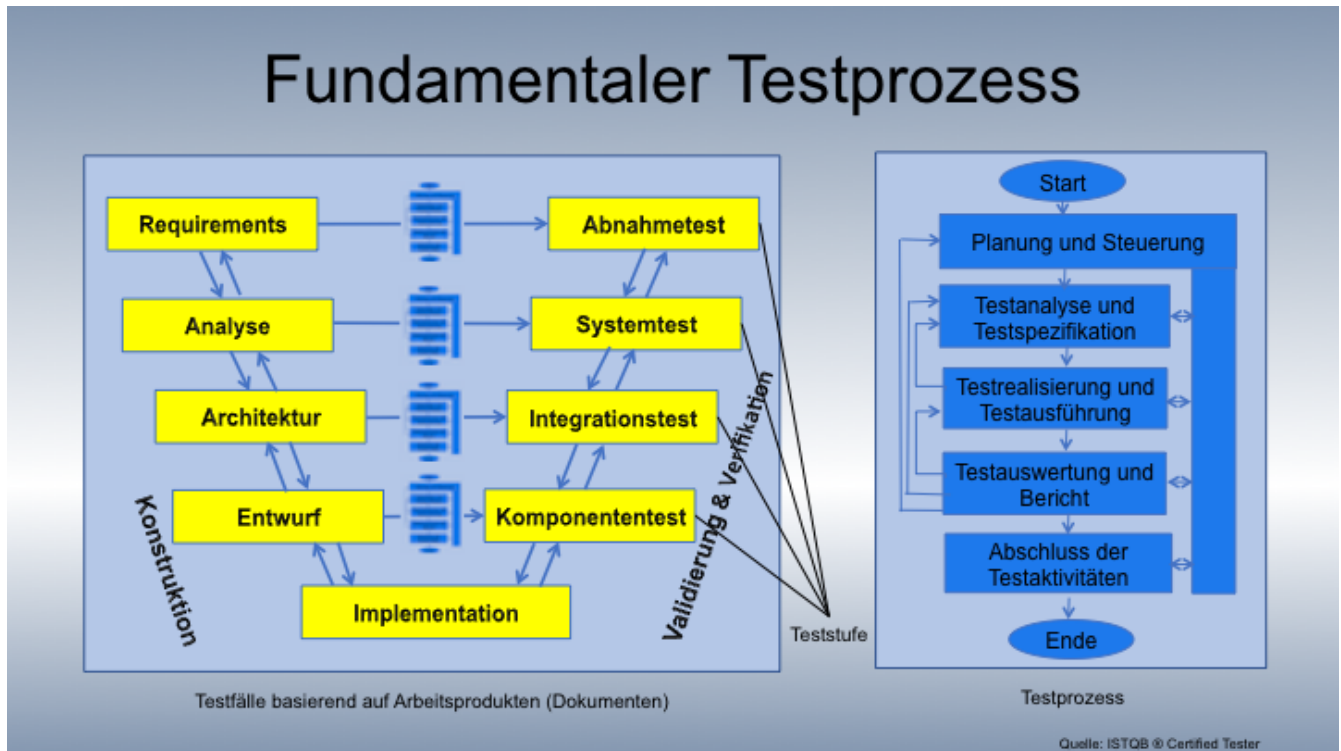
In welchen Teilbereichen der Software wird der Schwerpunkt liegen? Wie bewertet ihr das Risiko (wo liegt das Risiko) und wo liegen die Prioritäten im Testen?



# Testen im Software-Lebenszyklus

## Testplanung und Steuerung

Nach dem ISTQB-Standard stellt sich die Testplanung und Steuerung folgendermaßen dar:



Die ISTQB arbeitet für eine standardisierte Qualifikation von Software-Testern und fungiert als Zertifizierung-Instanz.

## Testziele

- Test auf Funktionalität
- Test auf Robustheit
- Test auf Effizienz
- Test auf Wartbarkeit
- Test auf unvollständige oder widersprüchliche Anforderungen

⇒ Ziel muss eine Continuous Deployment (Continuous Delivery + Installation)

## Teststufen

- Komponententest
  - JUnit
  - Kleinste Einheit
- Integrationstest

- Zusammenarbeit von Komponenten
- auch für Fremdsysteme
- Top-Down, Bottom-Up, Ad-hoc, Backbone
- Systemtest
  - aus der Sicht des Kunden
  - Prüfung ob Dokumentation und Anforderung stimmt
  - Produktionsumgebungsnahe Testumgebung
- Abnahmetest
  - Ist der Vertrag erfüllt? (vertragliche Akzeptanz)
  - Kann und mag der Benutzer damit arbeiten? (Benutzerakzeptanz)
  - Feldtest

Diese Teststufen sollte man **nicht** in einem Wasserfall-Modell sehen und **nacheinander** abarbeiten. Der Umfang der einzelnen Teststufen wächst mit dem Programm. Je früher das Programm in Feldversuchen getestet wird um so eher fallen diesbezügliche Probleme auf.

⇒ Agile Softwareentwicklung!

## Aufgabe

Analyse und Erweiterung der Tests im Beispielprojekt.

- ☐ Welche Tests gehören zu welchen Teststufen?
- ☐ Wie würden Tests auf den fehlenden Teststufen aussehen?

## Last- und Stresstest

- <https://www.testing-board.com/lasttest-tools-uebersicht/>
  - Silk Performer
  - Jmeter
- SOAP-UI für WebServices

## Aufgabe

- ☐ Erstellen und analysieren eines Lasttests

# Statische Tests

Bei statischen Testverfahren wird die Software nicht ausgeführt (wie bei den dynamischen Testverfahren) sondern analysiert. Dies kann automatisiert durch Software-Tools (Quelltextanalyse) geschehen und durch die menschlichen Denk- und Analysefähigkeiten (Reviews).

- Quelltextanalyse
  - <https://www.eclEmma.org/>
  - [https://de.wikipedia.org/wiki/Liste\\_von\\_Werkzeugen\\_zur\\_statischen\\_Codeanalyse](https://de.wikipedia.org/wiki/Liste_von_Werkzeugen_zur_statischen_Codeanalyse)
  - <https://www.sonarqube.org/>
- Review
  - IEEE 1028 (Standard for Software Reviews and Audits)

# Dynamische Tests

## BlackBox-Verfahren

- Komplettes Programm
- Zugriff "von Außen" ohne Kenntnis des Quelltextes
- Tools z. B.
  - SOAP-UI für WebServices
  - Selenium für GUI-Oberflächen
  - Insomnia für REST-Services

## Aufgabe

- ☐ Erstellen eines Selenium-Tests

## WhiteBox-Verfahren

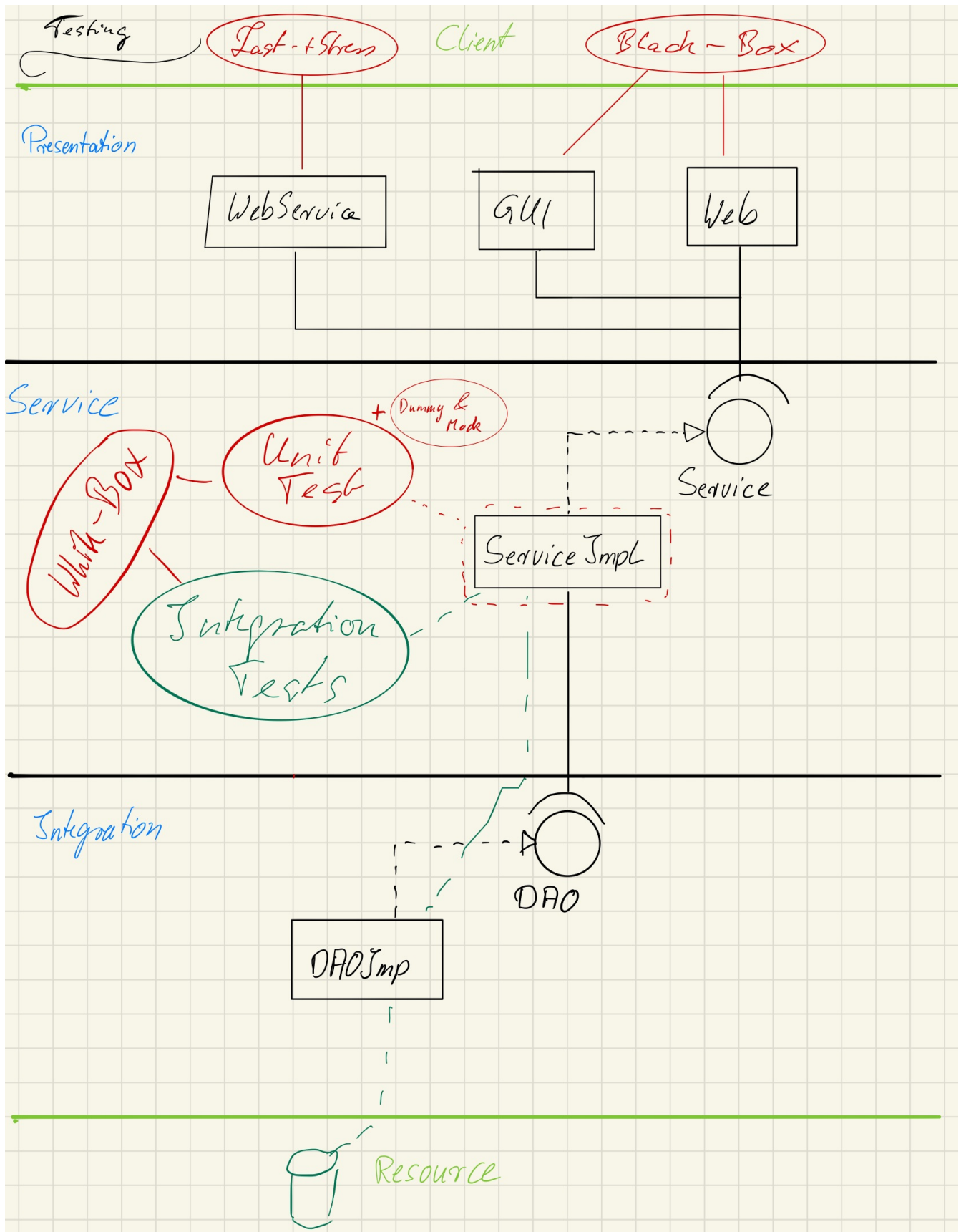
- Quelltext liegt vor
- Überprüfung der Wartbarkeit
- Möglichkeiten des Mockens
- Tools z.B.
  - JUnit-Tests
  - DB-Tests
  - Mocks mit EasyMock und Dummies

## Mocks und Dummies

Die Literatur unterscheidet einige Typen von Test-Doubles, sprich Objekten die als Ersatz der eigentlichen Objekte bei Tests eingesetzt werden um die Komplexität des Tests zu reduzieren.

Die gebräuchlichsten Typen sind Dummies und Mocks. Dummies sind sehr primitiv und geben immer den gleichen Wert zurück. Bei Mocks können Verhaltensweisen aufgezeichnet und am Ende verifiziert werden.

Weitere Typen sind "Fake-Objekt", "Test-Stub" oder auch "Test-Spy".



# Test-Driven-Development (TDD)

Regeln für TDD:

1. Produktiver Code darf ohne fehlschlagende Tests nicht geschrieben werden.

2. Nicht mehr Testcode als unbedingt notwendig ist, um einen Fehler anzuzeigen, darf geschrieben werden.
3. Nicht mehr produktiver Code als unbedingt notwendig darf für den Erfolg des fehlgeschlagenen Tests geschrieben werden.

Tests müssen sauber sein.

Seine Eigenschaften fasst das F.I.R.S.T.-Prinzip zusammen:

- Schnell ("fast"): Tests sollten schnell sein, sodass sie möglichst oft ausgeführt werden können. Tests, die viel Zeit in Anspruch nehmen, werden seltener ausgeführt. Fehler werden dann nicht sofort erkannt.
- Unabhängig ("independent"): Falls Tests voneinander abhängig sind, kann ein Fehler viele weitere anhängige Tests zum Scheitern bringen. Die Fehlersuche ist dann schwer. Unabhängige Tests können in beliebiger Reihenfolge und beliebig oft ausgeführt werden.
- Wiederholbar ("repeatable"): Tests sollten auf allen unterstützten Umgebungen ausführbar sein. Das heißt, sowohl auf der Entwicklungs-, der Staging- und der Produktionsumgebung funktionieren die Tests.
- Selbstvalidierend ("self-validating"): Das Ergebnis eines Tests sollte eindeutig sein. Entweder der Test war erfolgreich oder nicht. Mit Assertions kann man überprüfen, ob das Ergebnis korrekt ist.
- Zur richtigen Zeit ("timely"): Produktiver Code wird nur geschrieben, wenn es einen Test gibt, der fehlschlägt.

(aus Kai Spichale: Testwissen für Java-Entwickler.)

Weitere Hinweise:

- Red-Green-Cycle
- Tests schreiben bevor die Anforderung realisiert wird ⇒ Test richtet sich dann nicht nach der Realisation, sondern an der Anforderung.
- höhere Testabdeckung
- Code Katas z. B. <https://www.youtube.com/watch?v=9flsVKN4tZM>

## Aufgabe

- ☐ TDD für Passwortüberprüfung

# A. Anhang

## A1. Links & Quellen

### Zusätzliche Infos

- <https://www.istqb.org/>
- [https://de.wikipedia.org/wiki/Liste\\_von\\_Programmfehlerbeispielen](https://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen)
- <https://deanondelivery.com/product-managers-do-you-know-how-much-your-bugs-cost-72b6e36e7684>

### Bücher

- Basiswissen Softwaretest  
Andreas Spillner und Tilo Linz  
dpunkt.verlag  
ISBN 978-3-89864-642-0
- Testwissen für Java-Entwickler  
Kai Spichale  
Verlag: Software + Support  
ISBN-13: 9783868024692

### Tools

- JMeter <https://jmeter.apache.org>
- SOAP-UI <https://www.soapui.org/>
- Selenium <https://www.selenium.dev/>
- EasyMock <https://easymock.org/>
- JUnit <https://junit.org/junit5/>
- Jenkins <https://www.jenkins.io/>
- Insomnia <https://insomnia.rest/>

## A2. Stichwortverzeichnis

### @

10er-Regel, [3](#)

### A

Abnahmetest, [10](#)

Agile Softwareentwicklung, [10](#)

### C

Continuous Deployment, [9](#)

### D

Dummy, [11](#)

### I

Insomnia, [11](#)

Integrationstest, [9](#)

ISTQB, [9](#)

### J

JUnit, [9](#)

### K

Komponententest, [9](#)

Kosten, [4](#)

### M

Mocks, [11](#)

### Q

Quelltextanalyse, [11](#)

### S

Selenium, [11](#)

SOAP-UI, [11](#)

Systemtest, [10](#)

### T

Test-Doubles, [12](#)

## A3. Abkürzungsverzeichnis

### DB

Datenbank



**ISTQB**

International Software Testing Qualifications Board

**TDD**

Test-Driven-Development